Design and Implementation of a Multi-Gigabit NIC and a Scalable Buffered Crossbar Switch

Giorgos Kalokairinos, Vassilis Papaefstathiou, Aggelos Ioannou, Dimitrios Simos, Michail Papamichail, Giorgos Mihelogiannakis, Manolis Marazakis, Dionisios Pnevmatikatos, and Manolis Katevenis

Institute of Computer Science (ICS)
FOundation for Research and Technology – Hellas (FORTH)
P.O. Box 1385, Heraklion, Crete, GR-71110 Greece
{george, papaef, ioannou, simos, papamix, mihelog, maraz, pnevmati, kateveni}@ics.forth.gr

FORTH-ICS Technical Report, TR376-04-2006, April 2006

Abstract

High speed interconnection networks are a fundamental component of next-generation, scalable compute and storage systems. Although already a popular area of research, new application requirements and technology constraints impose new restrictions and present new opportunities for the design and implementation of interconnection networks. These include the need to exceed 10 GBit/s speeds, to further reduce host-related overheads, to support applications in a transparent manner, and to allow system scalability to large numbers of nodes.

This work presents the design and implementation of a multi-gigabit NIC and a scalable buffered crossbar switch that are currently used for research work in the area. The purpose of this work is to provide a detailed description of the architecture and its current implementation. We first provide an overview of the design, then we examine the prototyping infrastructure used, and finally we present the detailed NIC and switch implementation. Finally, we present the tests used for validating our implementation and we provide early performance results.

Design and Implementation of a Multi-Gigabit NIC and a Scalable Buffered Crossbar Switch

Giorgos Kalokairinos, Vassilis Papaefstathiou, Aggelos Ioannou, Dimitrios Simos, Michail Papamichail, Giorgos Mihelogiannakis, Manolis Marazakis, Dionisios Pnevmatikatos, and Manolis Katevenis

Institute of Computer Science (ICS)
FOundation for Research and Technology – Hellas (FORTH)
P.O. Box 1385, Heraklion, Crete, GR-71110 Greece
{george, papaef, ioannou, simos, papamix, mihelog, maraz, pnevmati, kateveni}@ics.forth.gr

FORTH-ICS Technical Report, TR376-04-2006, April 2006

Abstract

High speed interconnection networks are a fundamental component of next-generation, scalable compute and storage systems. Although already a popular area of research, new application requirements and technology constraints impose new restrictions and present new opportunities for the design and implementation of interconnection networks. These include the need to exceed 10 GBit/s speeds, to further reduce host-related overheads, to support applications in a transparent manner, and to allow system scalability to large numbers of nodes.

This work presents the design and implementation of a multi-gigabit NIC and a scalable buffered crossbar switch that are currently used for research work in the area. The purpose of this work is to provide a detailed description of the architecture and its current implementation. We first provide an overview of the design, then we examine the prototyping infrastructure used, and finally we present the detailed NIC and switch implementation. Finally, we present the tests used for validating our implementation and we provide early performance results.

1. INTRODUCTION

High speed interconnection networks are a central component of scalable compute and storage systems. Moreover, their importance is increasing as they are projected to play in increasing role in the design of all system components, and in particular both network and storage I/O as well as memory and CPU design. Current technology trends and application requirements impose new restrictions and present new opportunities for interconnect design.

Current application needs dictate moving to systems that can support higher than 10 GBits/s throughput and low response times. For instance, large-scale storage systems that require processing (even simple filtering) large amounts of (automatically generated) data, require moving significant volumes of data between disk, memory, and CPU. Furthermore, there is a need to scale to large numbers of system nodes, supporting both compute and storage nodes. To achieve these next generation interconnects there is a need to examine issues in both network interface controller (NIC) as well as switch architectures.

We believe that real prototypes are an excellent vehicle for studying architectural extensions. Many aspects of future architectures are the subject of current research. However, certain features, already present in today's systems, we believe will also be part of future systems, possibly at higher link speeds. For this reason, we have implemented a base NIC and switch architecture, encompassing many of these existing features that we believe will be part of future systems as well. In this work we present our prototype design and implementation, on which new architectural features will be implemented and evaluated.

The main purpose of this work is to present in detail the implementation of the RDMA-capable NIC and the buffered crossbar switch to support both future system extensions as well as system evaluation with real-life workloads. We implement our prototypes using state-of-the-art FPGA boards. The NIC supports asynchronous RDMA-write operations over multiple physical links. The buffered crossbar supports cut-through operation and credit-based flow control.

This paper is organized as follows. We first present an overview of the system and the base NIC and switch architectures in Section 2. Then we examine in detail the implementation of the NIC and the switch, presenting how these can be used by the operating system and user applications in Section 3. Section 4 briefly discusses how the implementation has been validated and we present preliminary performance numbers. Finally, in Section 5 we summarize our work.

2. SYSTEM OVERVIEW

Our system consists of NIC cards which are attached to host machines and the Buffered Crossbar Switch which provides a fast and efficient interconnect between the NICs of different machines. The switch allows the machines of the system to communicate with each other and benefit from the services offered by other machines. The NICs use the remote DMA (RDMA) technique to achieve the fastest possible transfer of large volumes of data and communicate with each other across the switch. The NICs also have extensive monitoring and debugging capabilities in order to measure the performance and export the state of the system. Figure 1 illustrates an overview of the system and a real photo of the system is shown in Figure 2.

The switch is a Buffered Crossbar switching fabric developed on a Xilinx Virtex II Pro FPGA with tens of RocketIOs and offers the connectivity between the NICs. The switch regulates the traffic and handles congestion with a backpressure flow control protocol between the switch and the NICs.

Each NIC card is developed on a Xilinx Virtex II Pro FPGA and is attached to the PCI-X bus of a host machine (Intel Xeon). The NIC card is also attached to the switch via the RocketIO network interface which provides the connectivity with the rest of the system. The PCI-X is the interface of the NIC with the host processor that runs the actual applications. This interface is used to initiate outgoing packets to the network and deliver incoming packets to the host memory. The RocketIO is a fast network interface and allows large data volumes to be exchanged with the other nodes of the network.

The system is used and tested with user initiated traffic but beyond this we have used a remote storage framework to allow more realistic traffic patterns. In this storage environment one of the machines hosts storage services and transparently shares a large volume of disk space among the other machines that belong to the network.

2.1 Network Interface

The prototypes of the NIC cards are developed on boards made by DiniGroup, namely the DN6000K10SC board [10], and feature a Xilinx Virtex II Pro FPGAs. Each development board has a rich collection of peripheral components such as SDRAMs, SRAMS and Flash memory while the Virtex II Pro device has 2 embedded PowerPC processors and 4Mbits of on-chip memory. Moreover the board has 4 RocketIO SMB interfaces and a 64-bit PCI-X interface.

The NIC design implements a 64-bit PCI-X interface which operates at 100MHz and can provide a maximum theoretical throughput of 760 Mbytes/sec to the host memory. The calculation of this throughput assumes a datum transferred in every cycle but actually there is the PCI-X protocol overhead. The DMA engine of the NIC has a fully functional 64-bit PCI-X Initiator with DMA capabilities and thus it can directly read or write to the host memory. The PCI-X Target interface of the NIC is also 64-bits and is used to accept commands from the host processor or deliver status and performance information.

Commands to the NIC from the host processor reach the PCI-X target interface and are placed in a memory mapped region which is called DMA Request Queue (Section 3.1.3). The request queue allows a maximum of 1024 pending operations with a maximum of 4Kbytes per operation (i.e. one operating systems page). The request queue also allows the user of the NIC to cluster operations and offers on-demand processing of every single operation. Another important feature of the request queue is the local notification which informs the processor about the completion of an operation. When the operation is completed, the NIC writes a value to a pre-specified memory address. The processor can then poll that memory address for the completion of a specific operation. Moreover the NIC offers the remote notification feature which can inform the processor for an incoming packet; this notification comes in the form of a level-triggered interrupt.

The NIC also implements a RocketIO network interface which is a high speed network interconnect offering 2.5Gbps of full-duplex network traffic. The RocketIO MultiGigabit Transceivers are offered by the Virtex II Pro devices and are used for the network transport.

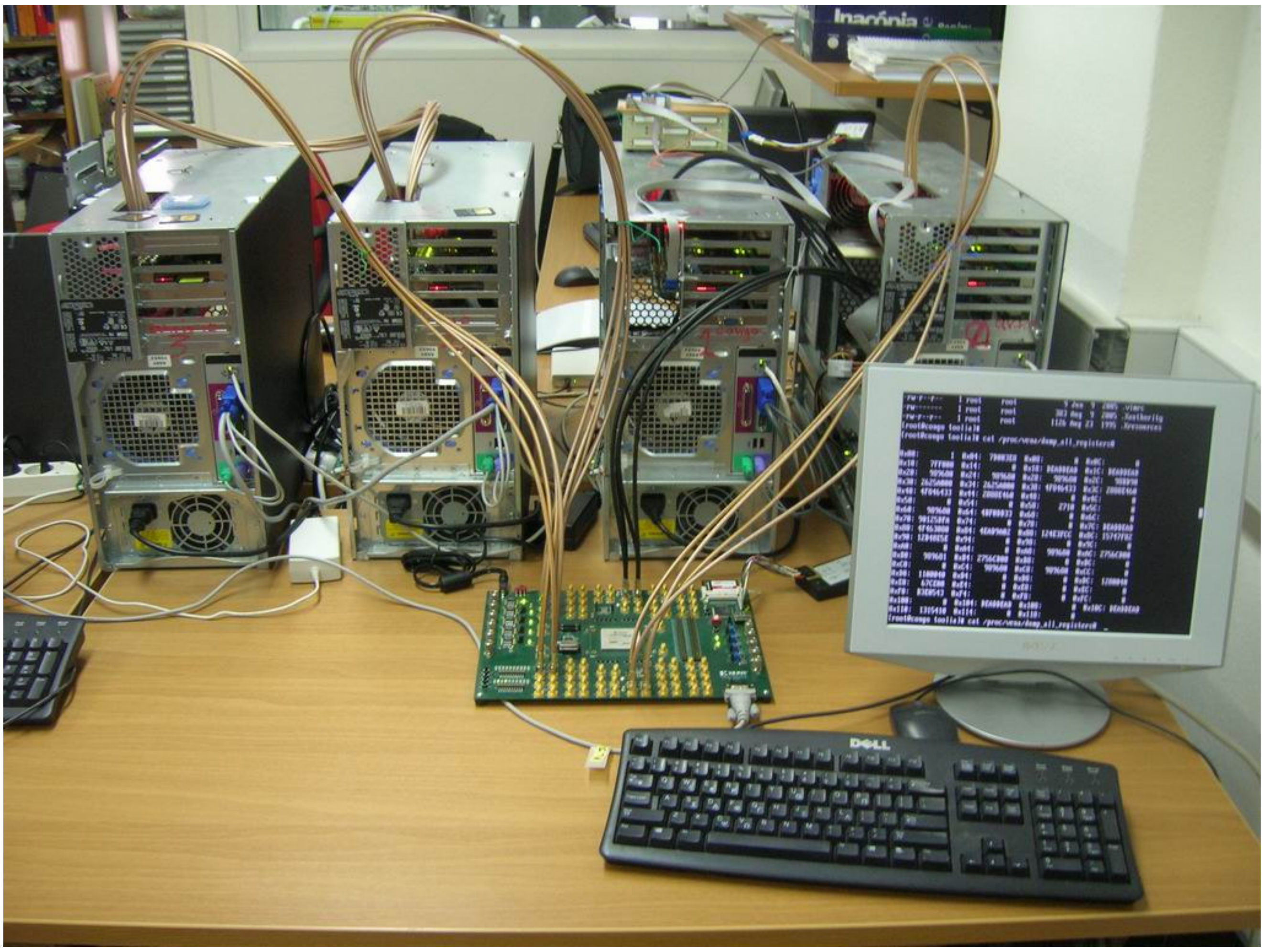

Figure 1: Picture of a 4-node system.

The NIC has two 8Kbyte network FIFOs associated with the incoming and outgoing network traffic. The outgoing FIFO is used as an elastic buffer but also as a classic input queue regardless of the destination – a virtual output queue implementation for every network destination is being implemented and we are ‘almost’ ready to use it and replace single outgoing FIFO. The incoming FIFO is also used as an elastic buffer but also covers the temporal uncertainties of the PCI-X DMA accesses. The credit based flow control mechanism is QFC-style and keeps state about every network destination (cross-point buffers in the switch). Credits and data share the same links and the credits are interleaved between the packets. A more detailed description of the NIC is provided in Section 3.1.

2.2 Buffered Crossbar Switch

The prototype of the switch is developed in a board made by Xilinx, namely the ML325 board [9], and features a Virtex II Pro FPGA with 2 embedded PowerPCs, an SDRAM and has 20 Rocket IO SMA interfaces and therefore is an ideal solution for a switch.

The switch that provides the connectivity in the system is a Buffered Crossbar and therefore the CICQ Architecture is implemented and RocketIOs serve as network interfaces. It is an 8x8 switch where only 4 ports are used until now. The switch prototype has 64 cross-point buffers where each of them is 2Kbytes. It has a single priority and has of course inherent ability to switch variable size packets while no segmentation and reassembly is needed. The operating frequency is 78.125MHz required by the RocketIOs and combined with the width of the internal paths is sufficient to fully utilize the network bandwidth. The switch implements a round robin scheduling policy with an output scheduler (OS) per output port and supports cut-through operation. The flow control is credit based (QFC-style) and is assigned to the credit schedulers (CS) that exist per input; these credits are interleaved between the packets. A more detailed description of the Buffered Crossbar Switch is provided in Section 3.2.

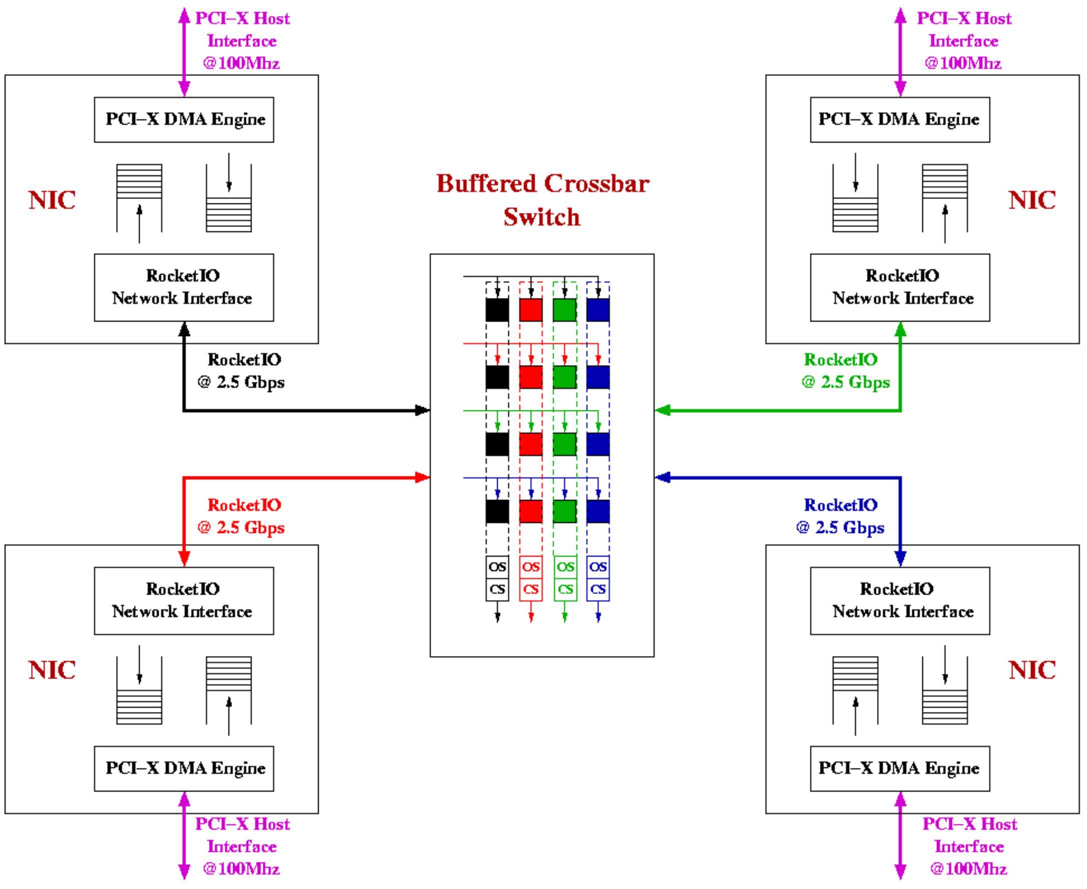

Figure 2: System Overview.

3. HARDWARE PROTOTYPES

In this section we present the hardware prototypes and the system that we built as part of current research. Our prototypes are developed on high performance FPGAs and offer numerous high speed interfaces that are used to create a rich and complex environment to host network services.

3.1 Network Interface

3.1.1 PCI-X DMA Engine Module

The block diagram of the PCI-X module is shown in Figure 3. The main NIC components are:

- The **Target** block implements the PCI-X target interface. The same block is used for the configuration space and therefore contains all the necessary configuration registers defined by the PCI-X standard. The DMA request queue, DMA operation registers and the performance counters/timers are updated and read through this target interface. These transactions can be 32 or 64-bits wide in burst or non-burst mode. The target block also supports dual address cycle. Moreover, one outstanding split completion is supported. Each such split completion can be 32 or 64 data bits wide. Finally, the target asserts the interrupt line upon receipt of the appropriate signal from the DMA Fifo block.
- The **DMA Initiator** consists of a PCI-X master interface while it also provides the full functionality of a DMA engine from/to the host's memory. It uses physical PCI addresses. It starts the DMA transfers according to

their order in the DMA request queue. These PCI-X transactions are multiples of 64-bits (ACK64_ is ignored) and can issue a dual address cycle access if the 32 most significant address bits are non-zero.

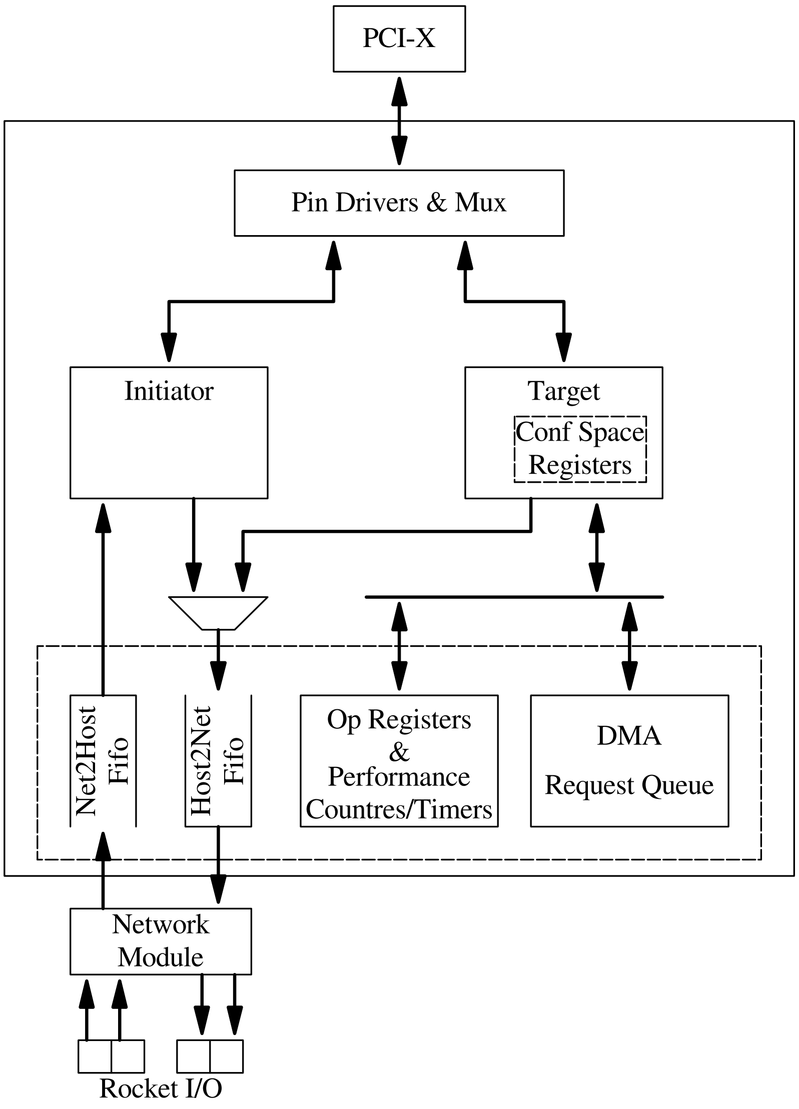

Figure 3: PCI-X DMA engine block diagram.

- The **Operation Registers & Performance Counters/Timers** block contains a control register, registers that store the PCI addresses for the remote and local notifications and several other performance counters. These registers are further explained in Section 3.1.4.1.
- The **Host to Net Fifo** is written by data acquired from the PCI bus, either from a block read transaction performed by the initiator, or by split completion transactions received by the target module. It also provides clock domain synchronization between the PCI and network clock. In the current configuration, the size of this fifo is 1023 words of 64 bits each. The FIFO size is not 1024 because of the design limitation relative to head and tail pointers.
- The **Net to Host Fifo** is written by the network module. It also provides clock domain synchronization between the PCI and network clocks. In the current configuration, the size of this fifo is 1023 words of 64 bits each.
- The **DMA Request Queue** holds up to 1024 outstanding requests, set by the host (later on they will be set by the on-chip CPU as well).
- Finally, the **Network Interface** block enables the communication of the PCI to the network.

3.1.2 PCI Configuration Space

All PCI functional devices must employ a block of 64 configuration 32-bit words for the implementation of their configuration registers. The next table illustrates the format of the configuration header region implemented in the NIC.

Address	**Name**			
0x00	Device ID		Vendor ID	
0x04	Status		Command	
0x08	Class Code			Revision ID
0x0C	BIST	Header Type	Latency Timer	Cache Line Size
0x10	Base Address Register 0			
0x14	Base Address Register 1			
0x18 to 0x30	Not implemented			
0x34	Capabilities List			
0x38	Not implemented			
0x3C	Not Implemented		Interrupt Pin	Interrupt Line
0x40	Application Interrupt Register			
0x44	Not implemented			
0x48	PCI-X Command		Next Capability	PCI-X Capability ID
0x4c	PCI-X Status			

- **Device ID:** This 16-bit value identifies the type of the device. Current value is: 0x17DF
- **Vendor ID:** This 16-bit value identifies the manufacturer of the device. Current value is: 0x1600
- **Status Register:** The status register tracks the status of PCI bus-related events. The next table explains the fields implemented in the NIC:

Status Register		
Bit Location	**Description**	**Action**
4:0 (LS)	Reserved	Hardwired to zero.
5	66MHz-Capable	Current value is: 1. The NIC is capable of operating at 66 MHZ.
6	UDF Supported	Current value is: 0. Device does not support UDFs.
7	Fast Back-to-Back Capable	Current value is: 0. Device does not support fast back-to-back transfers.
8	Data Parity Reported	Set by hardware if the master asserts the SERR_ (PCI line) and the parity error response bit in the Command Register is set.
10 : 9	Device Select Timing	Current value is: 0x02. The target device's decode speed is slow (decode time C).
11	Signaled Target Abort	Set by target whenever it terminates a transaction with target-abort.
12	Received Target Abort	Set by the master whenever its transaction is terminated by a target-abort from the currently addressed target.
13	Received Master Abort	Set by the master whenever its transaction is terminated due to a master-abort.
14	Signaled System Error	This bit should be set whenever a device generates a system error on the SERR_ PCI line.
15 (MS)	Detected Parity Error	A device should set this bit whenever it detects a parity error.

- **Command Register:** This 16-bit register provides basic control over the device's ability to respond to and/or perform PCI accesses. The next table explains the fields implemented in the NIC:

Command Register		
Bit Location	**Description**	**Action**
(MS) 15:10	Reserved	No action.
9	Fast Back-to-Back Enable	Currently hardwired to 0. Thus disabling fast back-to-back transfers (PCI-X specification requirement).
8	System Error Enable	When set the NIC can drive the SERR_ line.

7	Wait Cycle Enable	This bit is hardwired to zero (No stepping supported).
6	Parity Error Response	When set the device can report parity errors by asserting the PERR_ PCI line.
5	VGA Palette Snoop Enable	Currently hardwired to 0. Disable VGA Palette Snoop.
4	Memory Write and Invalidate Enable	The NIC DMA initiator does not support memory write and invalidate operations. This bit is ignored.
3	Special Cycle Recognition	The NIC do not respond to special cycles. This bit is ignored.
2	Master Enable	Enables the DMA initiator when set.
1	Memory Access Enable	When set, the device responds to PCI memory accesses.
0 (LS)	I/O Access Enable	Currently hardwired to 0. Disable I/O access.

- **Class Code:** This is a 24-bit read only register that defines the revision ID. Current value is: 0x020000
- **Revision ID:** This 8-bit value is assigned by the manufacturer to identify the revision number of the device. Current value is: 0x17
- **BIST:** The NIC does not implement built-in self-test so the value is currently hardwired to 0x00
- **Header Type:** Currently hardwired to 0
- **Latency Timer:** The Latency Timer defines the minimum amount of time, in PCI clock cycles that the master can retain ownership of the bus. The default value is 0x20 and the recommended value is 0xFE.
- **Cache Line Size:** This read/write configuration register specifies the system cache line size in 32-bit words. The recommended value is 0x80.
- **Base Address Register 0:** This register is written by the BIOS at boot time and contains the base address used to access the NIC RAM (SRAM and DRAM). Currently 27-bit address space is supported by the NIC.
- **Base Address Register 1:** This register is written by the BIOS at boot time and contains the base address used to access the DMA initiator registers (addresses 0x000 to 0x1FF) and the DMA request queue BRAM (addresses 0x200 to 0x3FF).
- **Interrupt Pin:** This register demonstrates which interrupt pin the device uses. The NIC uses the INTA_ PCI line so the current value is: 0x01
- **Interrupt Line:** The Interrupt Line register is read/write and is used to communicate interrupt line routing information's.
- **Application Interrupt Register:** This register is read/write and contains information about the interrupts generated from the NIC. The next table explains the fields of the register:

Application Interrupt Register Bits		
Bit Location	**Description**	**Action**
0 (LS)	Remote Interrupt Enable	If set, an interrupt is generated when a data packet from the network has arrived, provided that the corresponding operation code for this packet had the Interrupt bit set.
3:1	Reserved	Reserved for future expansion
4	Remote Interrupt Flag	Set by hardware when a data packet from the network has arrived, and its operation code had the Interrupt bit set.
7:5	3'b0	These bits are hardwired to zero.
15:8	8'b0	These bits are hardwired to zero.
23:16	Tail Pointer	Value of the request queue tail pointer.
23	1'b0	This bit is hardwired to zero.
30:24	Head Pointer	Value of the request queue head pointer.
31(MS)	1'b0	This bit is hardwired to zero.

- **Capabilities List Register:** The Capabilities Register points to the first item in the list of capabilities. This item is the PCI-X Command Register set therefore the Capabilities Register is hardwired to 0x48
- **PCI-X Capability IC:** This field is hardwired to 0x07 and identifies the Capabilities List as a PCI-X register set.
- **PCI-X Next Capability:** This field is hardwired to 0x00 and identifies that the PCI-X register set is the last entry of the Capabilities List.
- **PCI-X Command:** This register is read/write. The next table explains the fields of the register.

PCI-X Command		
Bit	**Description**	**Action**

Location		
0 (LS)	Data Parity Error Recovery Enable	If set the device will attempt recovery from data parity errors.
1	Enable Relaxed Ordering	If set the device is permitted to set the relaxed ordering bit in the requester attribute phase.
3:2	Maximum Memory Read Byte Count	This register sets the maximum byte count the device is permitted to use when the initiating a sequence with one of the burst memory read commands.
6:4	Maximum Outstanding Split Transactions	This register sets the maximum number of split transactions the device is permitted to have outstanding at any time.
(MS) 15:7	Reserved	These bits are hardwired to zero.

- **PCI-X Status:** This register is read only. The next table explains the fields of the register.

PCI-X Status		
Bit Location	**Description**	**Action**
2:0 (LS)	Function Number	Read only. Contains the Function Number.
7:3	Device Number	Read only. Contains the Device Number.
15:8	Bus Number	Read only. Contains the Bus Number.
26	64- bit Device	Hardwired to 1. The device supports 64 bits transfer.
17	133 MHz Capable	Hardwired to 1. The device is 133 MHz capable.
18	Split Completion Discarded	This bit is set if the device discards a split completion because the requester would not accept it.
19	Unexpected Split Completion	This bit is set if an unexpected split completion with this device's requester ID is received.
20	Device Complexity	Hardwired to 0 representing a simple device.
22:21	Designed Maximum Memory Read Byte Count	Hardwired to 2'b11. The maximum memory read byte count is 4096 bytes.
25:23	Designed Maximum Outstanding Split Transactions	Hardwired to 3'b000. The device supports only one outstanding split transaction.
28:26	Designed Maximum Cumulative Read Size	Hardwired to 3'b010 The maximum cumulative read size is 4K bytes
29	Received Split Completion Error Message	This bit is set if the device receives a split completion message with the split completion error attribute bit set.
(MS) 31:30	Reserved	These bits are hardwired to zero.

3.1.3 DMA Request Queue

The DMA Request Queue is a cyclic queue and uses two 2-port memories (implemented with BRAMs) in parallel with 2048 word x 32-bits configuration each and two pointers. We use two memories to allow 32-bit or 64-bits burst accesses to the queue via the PCI target interface. Each request is described with 2 entries in the queue. The first defines the PCI source address and the second the operation, word count and the remote host destination address as shown in Figure 4.

The Head Pointer points to the DMA request to be served and the Tail Pointer to the next free Entry of the DMA Queue (the head pointer follows the tail pointer even they wrap-around). The Tail Pointer is 10 bits (we can have up to 1024 pending Requests). The Head Pointer is 11-bits. The 10 most significant bits point to the currently served DMA Request and the least significant bit is used to copy each of the 2 request entries to the DMA engine. The DMA Request Queue is memory mapped and is accessible by the Host using the PCI target interface in the address range (Base Address Register 1 + 0x4000) up to (Base Address Register1 + 0x4FFF).

The structure of an entry in the DMA Queue is:

- The PCI Source address is used to define the physical host address where the data to be transmitted exist. These data will be read from the host memory with a DMA operation.
- The Remote Host Destination address is the physical address of the receiver where the data will be

transferred. Bits [31:0].

- The word count field indicates the size of the transfer in 64-bits words. The maximum size of each transfer is 512 words therefore 4096 bytes. Bits [41:32].
- The operation field indicates the features of the transfer. Bits [63:59]. These bits include a Remote Notification Flag (Bit [62]), Local Notification Flag (Bit [61]), Remote Interrupt Flag (Bit [60]) and a Start Flag (Bit [59]), which initiates the requests from the head pointer until the current request (clustering).

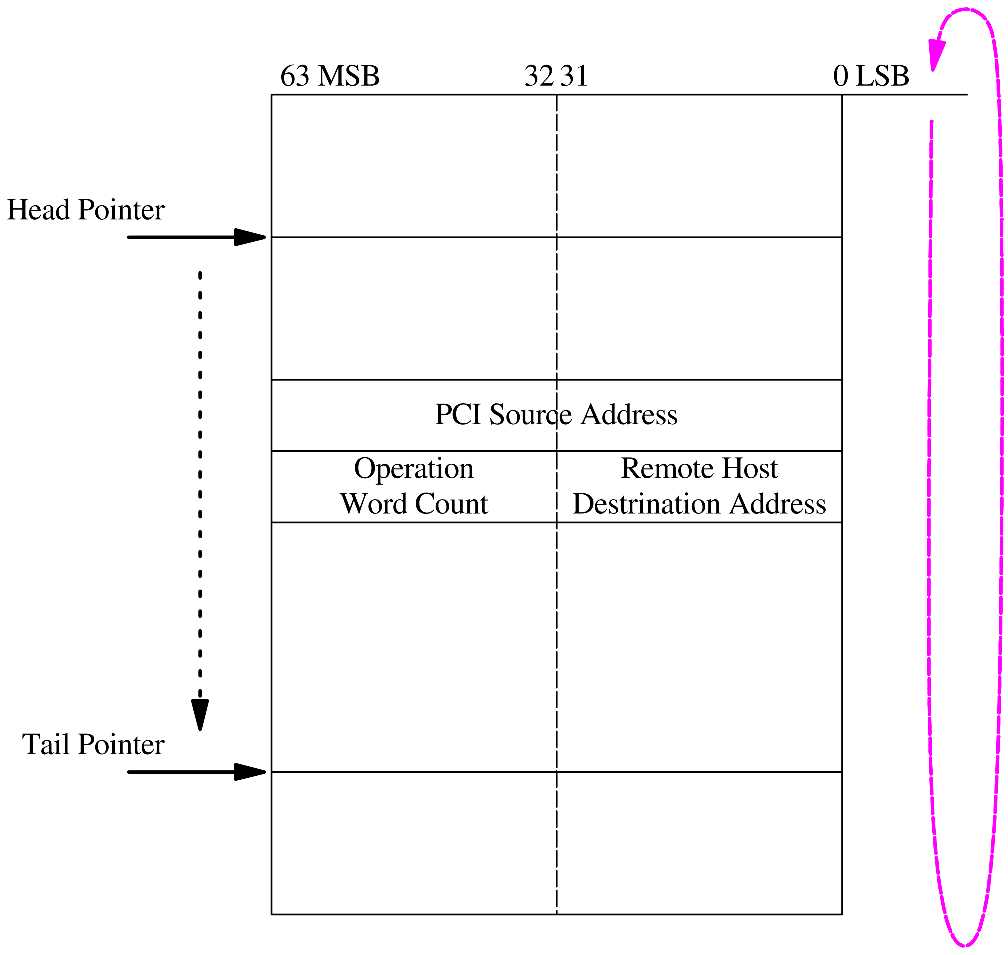

Figure 4: DMA request queue.

3.1.4 Operation Registers, Performance Counters, and Timers

This block contains several registers. These registers are accessible via the PCI target interface using the content of the *Base Address Register 1* and the address offsets given by the next table:

Address Offset	Register Name
DMA Operation Registers	
0x00	Control Register
0x04	Head Tail Pointers
0x08	Remote -Notification PCI Address Low
0x0C	Remote-Notification PCI Address High
0x10	Local -Notification PCI Address Low
0x14	Local -Notification PCI Address High
Network Module Counters	
0x80	Outgoing Net Cycles Counter
0x84	Incoming Net Cycles Counter
0x88	Rocket I/O Unknown Character Counter

Address Offset	Register Name
Performance Counters/Timers	
0x20	Host2NicDMAops Counter
0x24	Nic2HostDMAops Counter
0x28	Host2NicReq Counter
0x2C	Nic2HostReq Counter
0x30	Host2NicQWord Counter
0x34	Nic2HostQWord Counter
0x38	Host2NicDMAactive Timer
0x3C	Nic2HostDMAactiveTimer
0x40	Host2NicBusGranted Timer
0x44	Nic2HostBusGranted Timer
0x48	Host2NicDisc Counter

0x8C	Rocket I/O Loss of Sync Counter
0x90	Rocket I/O Parity Error Counter
0x94	Rocket I/O Tx Buffer Full Counter
0xA8	Outgoing Net Packets Counter
0xAC	Dequeued Words Counter
0xB0	Incoming Net Packets Counter
0xB4	Enqueued Words Counter
0xB8	Header CRC Error Counter
0xBC	Net Fifo Alignment Error Counter
0xC0	Net Backpressure Cycles Counter
0xC4	Host2Net Start Counter
0xC8	Net2Host Start Counter

0x4C	Nic2HostDisc Counter
0x50	IRQ Active Timer
0x54	IRQ Assertions Counter
0x58	Local Notification Counter
0x5C	Remote Notification Counter
0x60	Split Operations Counter
0x64	Split Duration Timer
0x68	Packet Body CRC Error Counter
0x6C	Packet Header Alignment Error Counter
0x70	Cumulative Timer
0x74	Host2NetfiFullError Counter
0x78	Nic2HostfiFullError Counter
0x7C	Interval Timer Max Value Register
0x80	Sample Counter
0x84	Sampling Memory Data

3.1.4.1 DMA Operation Registers

- **DMA Control Register:** This register is used for debugging purposes. The next table explains the fields of the register:

DMA Control Register [31:0]		
Bit Location	**Description**	**Action**
31:30	Net Loop Back	When set the net loop back is activated.
29:25		Not currently used.
24	Head and Tail Pointers Reset	When set it resets the head and tail Pointers
23:5		Not currently used.
4	Software Reset	If set by software, the fifos are cleared and the FSMs are set to their initial states.
3:1		Not currently used.
0	Initiator Enable	DMA Initiator enable bit.

- **Remote-Notification PCI Address Low:** The Remote-Notification PCI Address Low 32-bit R/W register contains the low order bits of the PCI address to which the remote notification message will be written.
- **Remote -Notification PCI Address High:** The Remote-Notification PCI address high 32-bit R/W register contains the high order bits of the PCI address to which the remote notification message will be written.
- **Local-Notification PCI Address Low:** The Local-Notification PCI address low 32-bit R/W register contains the low order bits of the PCI address used for the DMA finish notification.
- **Local -Notification PCI Address High:** The Local-Notification PCI address high 32-bit R/W register contains the high order bits of the PCI address used for the DMA finish notification.

3.1.4.2 Performance Counters / Timers

- **Host2NicDMAops Counter:** The Host to NIC DMA Operations counter is a 32-bit R/W counter incremented every time a DMA transfer from Host to NIC is completed (request queue FSM transition to state NoPndOp).
- **Nic2HostDMAops Counter:** The NIC to Host DMA Operations counter is a 32-bit R/W counter incremented every time a DMA transfer from NIC to host is completed (request queue FSM transition to state NoPndOp).
- **Host2NicReq Counter:** The Host to NIC Request counter is a 32-bit R/W counter incremented every time a PCI bus request from Host to NIC is completed (request queue FSM transition to state NoPndOp).
- **Nic2HostReq Counter:** The NIC to Host Request counter is a 32-bit R/W counter incremented every time a PCI bus request from NIC to host is completed (request queue FSM transition to state NoPndOp).
- **Host2NicQWord Counter:** The Host to NIC Quad-Word counter is a 32-bit R/W counter incremented every time a 64-bit word is transferred from the host to the NIC during a DMA operation.
- **Nic2HostQWord Counter:** The NIC to Host Quad-Word counter is a 32-bit R/W counter incremented every time a 64-bit word is transferred from the NIC to the host during a DMA operation.
- **Host2NicDMAactive Timer:** The Host to NIC DMA Active timer is a 32-bit R/W timer measuring the duration in clock cycles of the DMA transfers from host to NIC. The time interval starts when the REQ_ PCI signal is asserted and ends when the DMA is completed (request queue FSM transition to state NoPndOp).

- **Nic2HostDMAactive Timer:** The NIC to Host DMA Active timer is a 32-bit R/W timer measuring the duration in clock cycles of the DMA transfers from NIC to Host. The time interval starts when the REQ_ PCI signal is asserted and ends when the DMA is completed (request queue FSM transition to state NoPndOp).
- **Host2NicBusGranted Timer:** The Host to NIC Bus Granted timer is a 32-bit R/W timer measuring the sum of the time intervals during which DMA transfers from host to NIC take place. The time interval starts when the FRAME_ PCI signal is asserted and ends when the last data phase is completed (request queue FSM transition to state NoPndOp).
- **Nic2HostBusGranted Timer:** The NIC to Host Bus Granted timer is a 32-bit R/W timer measuring the sum of the time intervals during which DMA transfers from NIC to host take place. The time interval starts when the FRAME_ PCI signal is asserted and ends when the last data phase is completed.
- **Host2NicDisc Counter:** The Host to NIC Disconnect counter is a 32-bits R/W counter. The value of the timer is incremented every time a DMA transfer (Host to NIC) is disconnected due to a latency timer timeout.
- **Nic2HostDisc Counter:** The NIC to Host Disconnect counter is a 32-bits R/W counter. The value of the timer is incremented every time a DMA transfer (NIC to host) is disconnected due to a latency timer timeout.
- **IRQActivet Timer:** The Interrupt Request Active timer is a 32-bits R/W timer. This timer counts the PCI clock cycles that the interrupt request line INTA_ remains Active.
- **IRQ Assertions Counter:** The IRQ Assertion counter is a 32-bits R/W Counter. This counter is incremented at the interrupt line's negative edge.
- **Local Notification Counter:** The Local Notification counter is a 32-bits R/W counter. This counter is incremented with the completion of a local notification.
- **Remote Notification Counter:** The Remote Notification counter is a 32-bits R/W counter. This counter is incremented with the completion of a remote notification.
- **Split Operations Counter:** The Split Operations counter is a 32-bits R/W counter. This counter is incremented upon receipt of a split response.
- **Split Duration Timer:** The Split Duration timer is a 32-bits R/W counter. This counter is incremented at every clock edge between the receipt of the split response and the receipt of the first datum of the first split completion is supplied.
- **Packet Body CRC Error Counter:** The Packet Body CRC Error counter is a 16-bits R/W counter. The value of this counter is incremented every time a packet body CRC error is detected.
- **Packet Header Alignment Error Counter:** The Packet Header Alignment Error counter is a 16-bits R/W counter. The value of this counter is incremented every time an alignment error is detected
- **Cumulative Timer:** The Cumulative timer is a 32-bits R/W timer. The timer is activated by writing an initial value (eg: 0). Upon the assertion of the REQ_ PCI signal of the first DMA transfer the timer starts counting. Each subsequent read of this timer returns the time interval from the 1st request until the completion of the last DMA transfer *that has finished*. In other words, this timer is updated with the clock cycle count on the UpdtCnt state of the request queue FSM.
- **Host2NetFifoFullError Counter:** The Host to Net Fifo Full Error counter is a 32-bit R/W counter incremented every time a 64-bit word is transferred from the host to the Net during a DMA operation and is lost due to full Fifo enqueue.
- **Net2HostFifoFullError Counter:** The Net to Host Fifo Full Error counter is a 32-bit R/W counter incremented every time a 64-bit word is transferred from the Net to the host during a DMA operation and is lost due to full Fifo enqueue.
- **Configurable Sampling Facility:** The board offers a configurable sampling facility to estimate the inbound and outbound bandwidth. After specifying a sampling interval duration (expressed in PCI-X cycles) and a sample count, the board records in internal memory the number of 64-bit words transmitted and received during each of the sampling intervals. The internal sampling memory can hold up to 2048 samples. The collection of samples begins with the first DMA transaction after setting up the sample count parameter. The counts of inbound and outbound 64-bit words are accumulative. A utility program can read the sampling memory and compute the aggregate bandwidth (in MBytes/sec).
- **Interval Timer Max Value Register :** This 32-bit read write register holds the sampling duration expressed in PCI-X cycles.
- **Outgoing Net Packets Sample Counter :** This 32-bit read write counter holds the number of samples to be collected.
- **Sampling Memory Data:** For each sampling interval, this memory module holds two 32-bit counts, corresponding to the number of 64-bit words transmitted and received during each of the sampling intervals. This memory module can hold up to 2048 samples.

3.1.4.3 Network Counters

The next 19 counters are used for debugging purposes and are read clear counters. The software can read their value performing a load operation to the corresponding offset or clear all the counters simultaneously performing a store operation to the offset 0x80.

- **Outgoing Net Cycles Counter:** The Outgoing Net Cycles Counter is incremented in every network clock cycle the network block transmits data or credits.
- **Incoming Net Cycles Counter:** The Incoming Net Cycles Counter is incremented in every network clock cycle the network block receives data or credits
- **Rocket I/O Unknown Character Counter:** The Rocket I/O 0 Unknown Character Counter is incremented every time the Rocket I/O 0 detects an Unknown Character.
- **Rocket I/O Loss of Sync Counter:** Rocket I/O 0 Loss of Sync Counter is incremented every time the Rocket I/O 0 detects a Loss of Sync.
- **Rocket I/O Parity Error Counter:** The Rocket I/O 0 Parity Error Counter is incremented every time the Rocket I/O 0 detects a Parity Error.
- **Rocket I/O Tx Buffer Full Counter:** The Rocket I/O 0 Tx Buffer Full Counter is incremented every time the Rocket I/O 0 is forced to discard a byte due to the Tx buffer being full.
- **Outgoing Net Packets Counter:** The Outgoing Net Packets Counter is incremented every time a packet is send.
- **Dequeued Words Counter:** The Dequeued Words Counter counts the numbers of words send by the network block.
- **Incoming Net Packets Counter:** The Incoming Net Packets Counter is incremented every time a packet is received.
- **Enqueued Words Counter:** The Enqueued Words Counter counts the number of words received from the network block.
- **Header CRC Error Counter:** The Header CRC Error Counter is incremented every time the network module detects a header CRC error.
- **Net Fifo Alignment Error Counter:** The Net Fifo Alignment Error Counter is incremented each time incoming network data are misaligned.
- **Net Backpressure Cycles Counter:** The Net Backpressure Cycles Counter represents the network clock cycles the network module was forced to wait due to lack of credits (backpressure).
- **Host2Net Start Counter:** The Host2Net Start Counter counts the assertions of the Host2Net Start signal.
- **Net2Host Start Counter:** The Net2Host Start Counter counts the assertions of the Net2Host Start signal.

3.1.5 Link Interface

The network interface module utilizes mainly two FIFOs that are used so as to store the outgoing (Host2NicFIFO) and incoming (Net2HostFIFO) data to/from the network modules of the NIC. The size of the FIFOs is 1023 68-bit words and they also provide four control signals, namely empty, almost empty, full and almost full, along with the actual utilization of the FIFO (i.e. the number of entries currently occupied). The words are 68-bits since the 64 LS bits are used for the data and the other 4-bit carry control information such as start of packet (bit 65), end of packet (bit 66) and the next 2 bits are reserved for word enables to show which 32-bit data words are valid. Those FIFOs are connected to the network modules using the interface specified in the following table.

“PCI-block to Network” Module Interface			
Pin Name	**Type**	**Size (bits)**	**Description**
Host2NetFiDt	Output	68	Host to network FIFO data out.
Host2NetFiDeq	Input	1	Network to host FIFO dequeue.
Host2NetPReady	Output	1	Host to network packet ready.
Net2HostfiDt	Input	68	Network to host FIFO data in.
Net2HostfiEnq	Input	1	Network to host FIFO enqueue.
Net2HostfiSt (Synchronized with the Network Module)	Output	9	Network to host FIFO status: Number of 64-bit words contained in the net to host FIFO.
Net2HPReady	Input	1	Net to host packet ready. This signal indicates that the incoming packet is in the net to host FIFO.
Net2HHReady	Input	1	Net to host header ready This signal indicates that the header of the incoming packet is in the net to host FIFO.
NetPcktError	Input	1	Net packet error. This signal indicates the network detects a header error.
Loop	Output	2	Network Loop Back.

			00: Normal operation. 01: Not supported. 10: Not supported.

On the outgoing path, Host2NetPReady is asserted every time a full packet is enqueued into the Host2NetFIFO and thus it is ready to be transmitted over the network. The Host2NetPReady signal is synchronous to the net clock and is asserted for one clock cycle. Then the network module, when it decides that it can process the packet, is responsible for raising the Host2NetFiDeq signal so as to start the dequeue from the Host2NetFIFO. The Host2NetFiDtout is the 68-bit data bus that is connected to the output of the Host2NetFIFO. There is also the Host2NetFiSt status bus that is synchronized with the network module's clock and is used so as to allow the network module to read the status of the Host2NetFIFO.

A network packet consists of the header part, the data part (body) and the CRC part. The header part is the first 68-bit words of the packet and is explained in the table below. Their four most significant bits identify all such parts. A value of zero represents a data word, a value of one a packet header and a value of two a body CRC word. The CRC we use is the popular CRC32-Ethernet / AAL5 which is 32-bits and its polynomial is:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x^{1} + 1$$

Header Part Fields		
Bit Location	**Description**	**Action**
(MS) 67:64	Word Identifier	The value of these bits is 4'b0001 for the header part
63	Packet Type Identifier	This bit is set to 0
62:56	Node ID	128 nodes are supported
55	Operation Code	This bit is set to 0
54		Remote Completion Notification Flag
53		Local Completion Notification Flag
52		Remote Interrupt Flag
51		This bit is set to 1
50:42	Not implemented	Hardwired to zero
41:32	Packet Size	This field contains the payload size in 64 bit words
31:0 (LS)	Designation Address	This field contains the destination PCI Address

3.1.6 RocketIO Module

The Network Interface (NetIF) Module is the part of the design that enables the communication of the rest of the FPGA modules with the network. This interface is implemented with the Xilinx RocketIO transceivers. The communication with the rest of the fpga is accomplished through a very simple interface, which mainly uses asynchronous fifos (as different clock domains are crossed), and just a few handshaking signals.

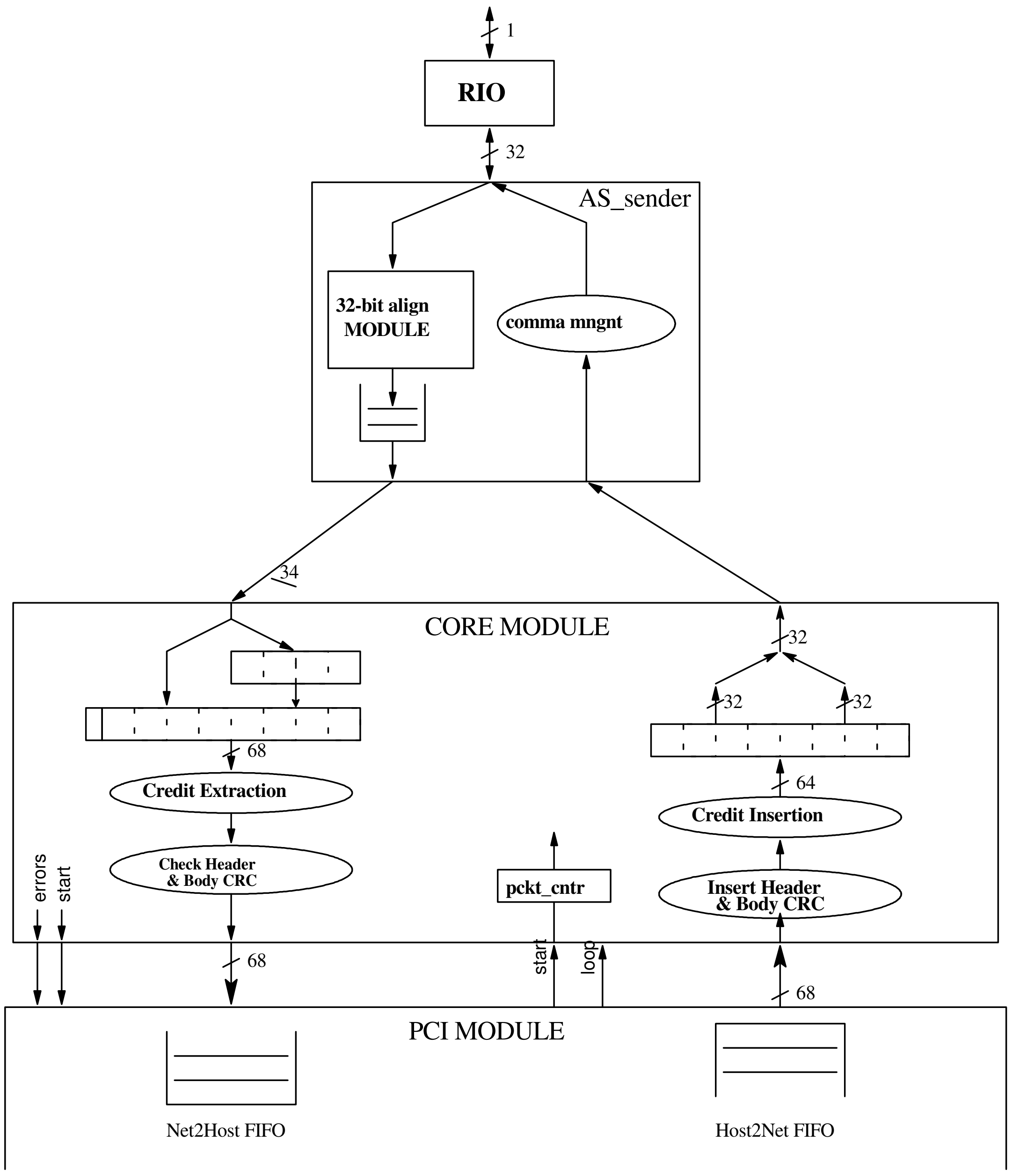

Figure 5: Network interface module block diagram.

The basic structure of the NetIF is outlined in the block diagram of Figure 5. We will carry on explaining the functionality of the NetIF as we look at this block diagram. At the bottom of the diagram we see a small part of the PCI module, which is the one directly linked to the NetIF. For each direction, incoming and outgoing, the PCI module deploys a separate asynchronous fifo. The width of the fifo is 68 bits, from which 64 are used for data, and the remaining 4 bits for tagging (flags for several purposes). This width of each of the two fifos is directly linked to the data format exchanged among the two modules. The data to be exchanged is formed into packets, and is then send to the NetIF. The NetIF on the other hand, takes that packet, sends it to the network after applying the needed modifications and additions. At the receiving side, it takes a packet, and after striping any network overhead, gives it to the PCI module (or any other receiving module). Explanations on the packet format can be found in Section 3.3.

Let us first describe **the transmitting side** of the diagram. After the PCI module has inserted a complete packet into the Host2Net fifo, it raises the start signal for a single clock cycle. The NetIF knows that now at least a single full packet resides in the fifo. As many packets can simultaneously reside in the fifo, it is the NetIF's responsibility to keep track of the number of packets still residing therein. This of course must be done correctly even if a packet transmission is active at the time a "start" is signaled. This functionality is carried by the module tagged pckt_cntr (packet counter) in the diagram. Any time a full packet resides in the fifo, the NetIF has the responsibility to read it, and safely transmit it to the network. To have additional error checking capabilities, the NetIF adds a header CRC while transmitting, right after the header is send. This is a CRC calculated on the header bits, which are the most critical data of a packet. Details on the CRCs used can be found in the table below. As seen in the diagram the data are sent to the network through the RocketIO transceiver. The RocketIO can reach up to 3.125 GBaud (2.5 Gbps). We use them at this maximum rate. The AS_sender module, which is directly linked to the RocketIO, adds synchronization information to the link, at the time periods that no useful network traffic exists.At **the receiving side**, each AS sender has a small synchronization fifo. This is used in order to form 64-bit words, by combining the two 32-bit words of each link. This is needed, as the exact arriving time of the bits at each link may and will differ, and so the 32-bit words are not presented always at the same clock cycle from the RocketIO to the AS_sender. The fifo then serves for removing this uncertainty. Before this, the problem of the alignment must be solved.

As the RocketIO sends data through a serial link, the **deserialization process** at the receiving side can cause misalignment of the bytes. That is the bytes may be found shifted in the received 32-bit word. This problem can be overcome by using a separate circuit for each link, which uses two consecutive words to form a valid one. This module is the rcv_align module. This is responsible for taking the misaligned incoming stream, extract the SOP and synchronization information from the link, and deliver the packet in its original from. The data are also accompanied by a start flag, in order to pass delimiting information to the core module.

	# bits	**Function**
Header CRC	16	Polynomial: (0 5 12 16), data width: 64, initial CRC value: 16'hFFFF first serial data bit: D[63]
Body CRC	32	polynomial: (0 1 2 4 5 7 8 10 11 12 16 22 23 26 32), data width: 64, initial CRC value: 32'hFFFF_FFFF first serial data bit: D[63]

The whole information (data and flags) are enqueued in the small fifo mentioned earlier, and through it are delivered to the core module. After the data are restored to the state they were sent, the **incoming data processing** can be made easier. In the beginning we check for **credits**, and after zero or more credits, zero or one data packets can follow. The **data packet** starts with the header, followed by the header CRC. The header CRC is checked and if it is found to be wrong the packet received is not passed to the PCI module. If it is correct, it is send, but only after it is correctly added to a 68-bit word, and the flag bits (the 4 MS bits) of each packet word are accordingly set. At the end of the packet the body CRC is checked. Correct or not, the packet has already been enqueued, so the result of the body CRC is just passed across to the other module. On a NIC, both the header and the body CRC are striped from the packet before it gets delivered. However on the slightly differentiated version for the crossbar switch, the body CRC is delivered. Finally let as mention that the output labeled "errors" represent a number of signals that inform the other module of some common errors (header and body CRC error, alignment error). The last data word is accompanied by an eop flag (bit 65).

3.1.6.1 Logic FSMs

Figure 6 depicts the FSM that serves the outgoing path (PCI module to network). This diagram just includes the states and the transitions, but bare in mind that quite excessive checking is included in most of the states, in order to accommodate for proper traffic manipulation. This also stands for the outgoing path FSM, which we will describe shortly.

As we see in Figure 6 the FSM starts from the idle state, and stays there as long as no traffic exists. When a full valid packet is found in the Host2Net fifo, the circuit starts sending the module. As a complete packet is enqueued before this process starts, the FSM can handle the whole packet non-stop, as the difference between the frequencies of the clocks for the two different sides does not matter. First the SOP is send to the network. This is not included in the packet, and is added as a network delimiter to a new packet. Then the header is read and sent. At the next cycle, a header CRC is calculated and is sent over the network. Again, this is not part of the original packet, and will be used by the receiving network interface, and will then be striped off. Then the data is sent. The circuit has two ways to double check when the data ends. One is the size that was found in the header, and the other is the flag at the last 68-bit word that signals the last word. At the end the body CRC, which was calculated all the way through the payload data transmission, is given. Notice that in order to enhance performance, the next state of "body_CRC" is not only the "idle", but it can also be "sop". This is done when a new packet is ready at the time the current packet is being sent. At this circumstance, we can immediately move to the “sop” state, and avoid the many unneeded commas.

Figure 7 depicts the FSM that serves the incoming path (network to PCI module). It starts with the start state. This state has the responsibility to find a start of packet from both of the links, which have been already properly aligned. This state also manages the incoming credit management. Transmissions can consist only of credits, or credits and a data packet, in which case one or more credits precede the packet. If errors occur, the erroneous packet is discarded, and a new packet arrival is awaited. If the process succeeds, we move on to the “headerCRC” state along with the enqueing of the header to the Net2Host FIFO. However, this is not done before this state, the “header_CRC”, receives the Header CRC, and checks its correctness. If it is valid the packet will be enqued to the fifo, as the FSM advances to the next state. However, if it is found to be wrong, the enqueuing is deactivated, and the rest of the states are used to just discard the packet. At this time, the NetIF signals to the PCI module that a packet has arrived, and so the later can start dequeuing it from the Net2Host fifo. Although this is not absolutely true, it will not result to any errors, as the network runs with a faster clock compared to the PCI module (78.125MHz vs. 66.67MHz). This way cut-through is implemented in the network-to-host stream. In later versions that use PCI-X with faster clocks (100MHZ-133MHz) this is not implemented. Instead a mechanism which is aware of the two frequencies (through user definitions) calculates a suited time in order to signal start and thus begin cut-through as soon as possible. When all the data are enqueued to the Net2Host fifo, the EOP flag will be received along with the last data word. Then the body_CRC will be checked and the FSM will get to the “start” state.

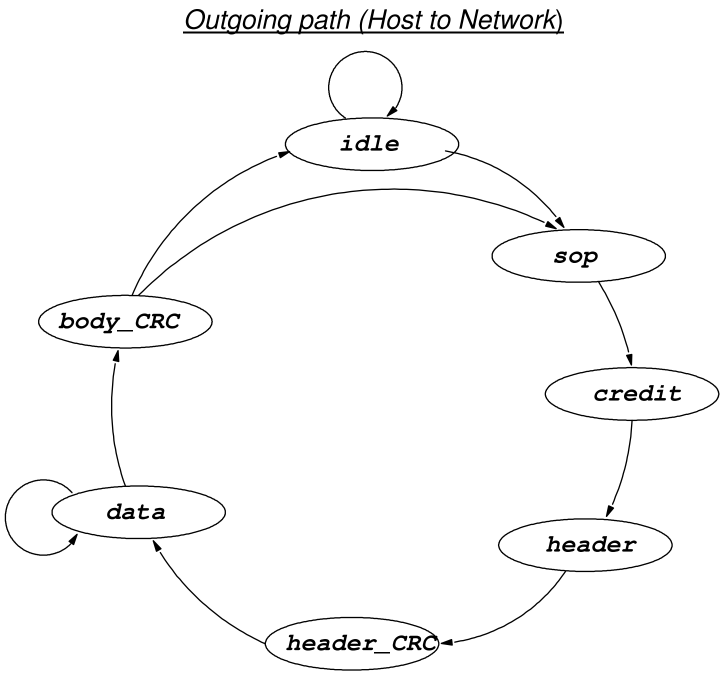

Figure 6: Host to network FSM.

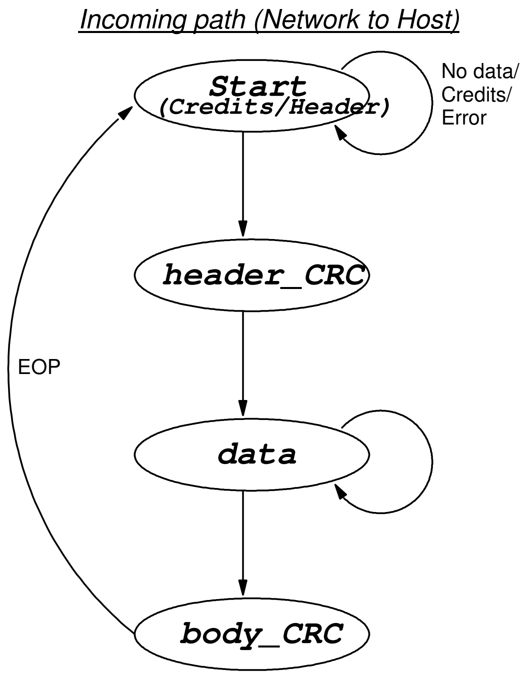

Figure 7: Network to host FSM.

3.1.6.2 Round Trip Time (RTT) Considerations

The time that is needed for information to travel from the transmitter to the receiver and again back at the transmitter (the RTT) is a quite important parameter of a system. For this system we have measured the RTT with two different manners: theoretically and through lab testing. In this way we have been able to compare the two results and see if they agree, in order to have a more reliable RTT value. For the lab testing we have added some logic in our original design that performs the following measurement. It counts cycles spent from the beginning of a packet transmission up to the point that an updated credit gets received. In this way we are sure that the packet has started being delivered at the receiving host, and after some time, a credit arising from that packet delivery is formed and sent back to the transmitter. Different experiments have been carried, with different packet sizes to make sure of the correctness for the lab measurement. The resulting RTT is about 90 network side clock cycles (78,125MHz clock) plus a number of cycles equal to packet-size/4. This last term is due to the cut-through latency, which delays a packet from being delivered from the NIC to the receiving host. The “/4” is for the latest system that uses PCI-X at 100MHz, and thus is capable of starting delivering the packet to the host, after the first quarter has arrived from the network.

$$RTT = (90 + \frac{packet\_size}{4})cc$$

This number conforms to the theoretical approach and thus the lab measurement seems quite robust. Briefly referring to the theoretical approach, we mention that the 90 clock cycles mostly involve RocketIO-to-network and vice-versa (about 45cc), credit granularity (32cc) and NIC-to-PCI latency (about 5cc).

3.1.6.3 RocketIO Transceiver Instantiation

In this paragraph we will take a look at some detail concerning the way the RocketIO is used by our design. The reader should be familiar with Xilinx's RocketIO transceiver. A reference for this is "RocketIO Transceiver User Guide" by Xilinx. The GT_CUSTOM primitive was used, as it is the most flexible, allowing the most user modifications. In the current fpga, xc2vp40, the X1Y1 and X2Y1 MGT locations at the top of the fpga were used. The COMMA character was chosen to be the default one, K28.5 ('hBC), and K27.7 ('hFB) is used as the SOP character. BREFCLK2 was used as the clock input, as the clock we provide is of a frequency greater than 2.5GHz. Actually the crystal mounted on the board gives the higher accepted frequency, which is 3.125GHz. BREFCLK2 was used instead of BREFCLK, as the crystal output on the board enters the fpga through pins H17 and J17, and so the internal clock routing of the MGTs obligates us to use BREFCLK2. The USERCLKs are produced from the clock given to the MGTs, with the help of a DCM. The USERCLK is given the MGT clock divided by two, as we use a user datapath width of 32 bits. The USERCLK2 is the USERCLK shifted by 180 degrees. As far as clock recovery is concerned, CLK_CORRECT_USE is set to true, to allow correction. CLK_COR_REPEAT_WAIT is set to 0 and CLK_COR_KEEP_IDLE is set to false, to allow the maximum capability for clock correction. The RX_BUFFER is set to true to also aid this cause. CLK_COR_SEQ_1_1 is set to the COMMA character, and CLK_COR_SEQ_LEN is set to 1, to allow single COMMA to be the sequence that is allowed to be removed. The CRC insertion is set to inactive, as the header CRC insertion and checking is done through user designed circuits. One last thing concerning the RocketIO attributes is that the TX_DIFF_CTRL is set to 800 instead of the default value 500, and TX_PREEMPHASIS is set to 3 instead of the default value 0. These two values are based on lab testing, as the default values lead to a very high error rate. Note however that even more moderate setting (e.g. 500 and 2 respectively) could probably lead to a working set. Finally, as far as loopback is concerned, the core_AS module has a 2-bit input that accepts the loopback mode. The coding of the modes is the one given by the RocketIO specifications, and the user must be cautious to drive this input at the correct value at all times.

3.2 Buffered Crossbar

In this section we present the organization and the building blocks of the buffered crossbar switch regarding the prototype implementation in a high performance FPGA. This implementation is based on the research results presented in [5] and the initial design [8] that proved the feasibility of a 32x32 buffered crossbar in modern ASIC technology. The current report focuses on porting the latter design in a high end FPGA environment which can support millions of transistor logic, several Mbits of on-chip memory and many multi-gigabit network links.

3.2.1 Buffered Crossbar Organization

The buffered crossbar switch is implemented in a development board made by Xilinx. We use the ML325 Characterization Board [9] with features a Virtex II PRO XC2VP70K device and has 20 RocketIO SMA interfaces on the board. The FPGA device has 74K logic cells and 6Mbits of on-chip memory in 328 embedded Block RAMs (BRAMs) of 18Kbits each. Moreover, its 20 RocketIO hard-blocks operate at 156.25 Mhz and each one provides 2.5Gbps of full duplex network throughput. The internal interfaces of the RocketIOs operate at 78.125Mhz on a 32-bit datapath, hence this fact sets the frequency limit of the crossbar. Since the RocketIOs are placed in both the top and bottom side of the FPGA, the manufacturer demands two clocks deriving from different crystals and therefore we consider two clock domains for the switch. Those features of the board enable us to develop a prototype of the buffered crossbar with decent size, performance and real life characteristics.

The architecture of the crossbar is based on the work of [5] and [8] where all the related details are discussed. We have implemented an 8x8 buffered crossbar, with 32-bit datapath, on the FPGA whereas we managed to achieve a 10x10 configuration utilizing 65% of the FPGA logic. This configuration seems to be the practical limit for the current FPGA device since there are too many wires to be routed.

For the 8x8 configuration we have 64 cross-point buffers with 2Kbytes and 8 RocketIOs for the network interfaces. In our custom network protocol the packet headers/trailers are 16 bytes and the minimum payload size is 8 bytes (one 64-bit word of NIC's PCI-X). Consequently the minimum size of a packet is 24bytes which is translated in 6 clock cycles in our 32-bit datapath. We also defined the maximum payload size to 496 bytes hence the maximum packet size is 512 bytes (128 cycles). This maximum size is an effect of the size of each cross-point buffer and the round trip time (RTT) of the network, as explained in [5].

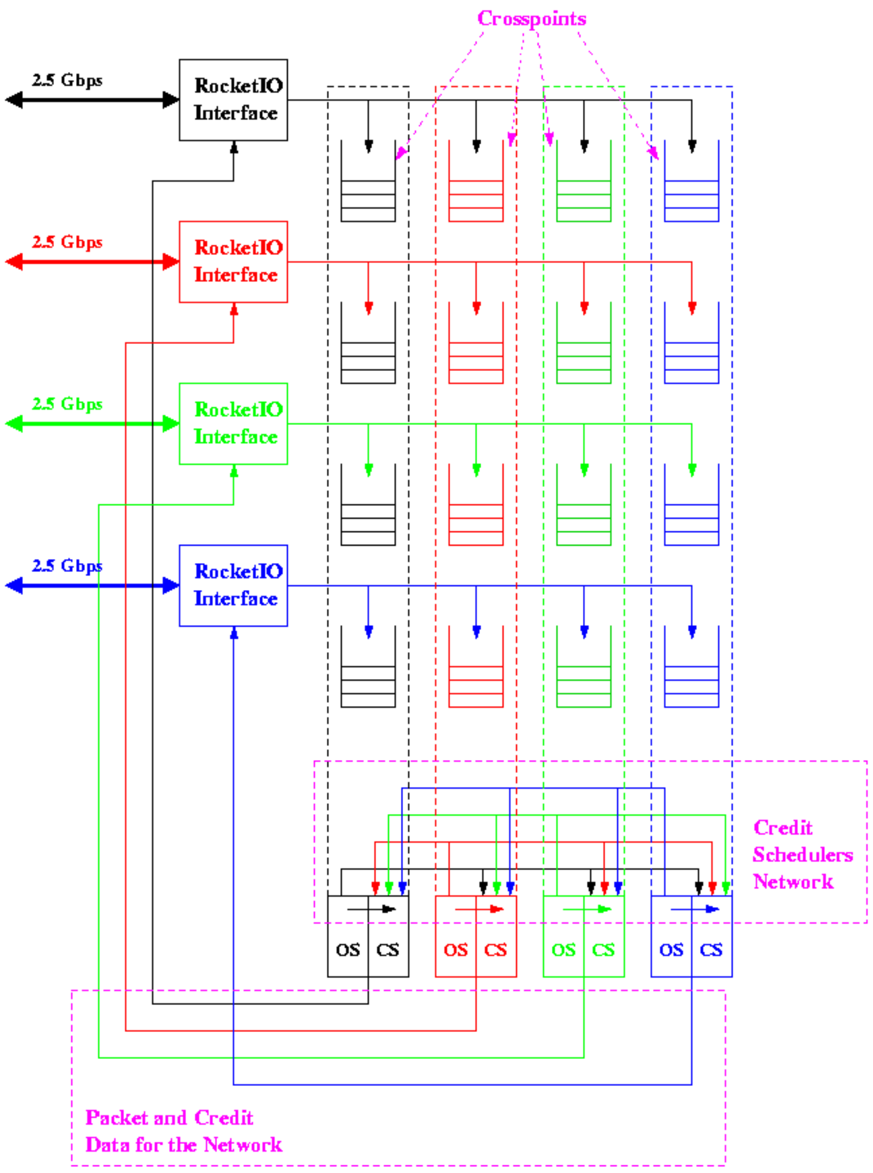

Figure 8: Buffered crossbar internal architecture.

The building blocks of the switch are:

- Cross-point buffers and associated logic.
- Output schedulers (OS) per output.
- Credit schedulers (CS) per input.
- Per link RocketIO interfaces.

The block diagram of a 4x4 buffered crossbar is shown in Figure 8 and a real photo in Figure 9.

3.2.2 Cross-point Buffers

Each cross-point buffer is a 2Kbyte dual ported show-ahead FIFO in a 512x32 configuration and utilizes 1 BRAM for each buffer. The logic of the cross-point receives the packets and the associated flags (start of packet and end of packet) from the network interface and enqueues them to the memory. When a new packet starts entering the cross-point buffer then the logic informs the output scheduler (OS) of the crossbar column through a synchronizer circuit; remember that there are two clock domains. Additionally, the cross-point buffer is able to dequeue a 32-bit word in every cycle when the OS requests it.

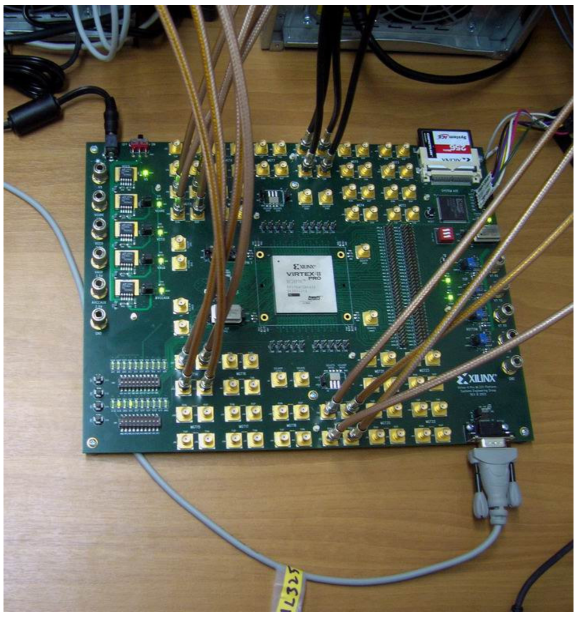

Figure 9: Picture of switch prototype board.

3.2.3 Output Scheduler (OS)

The output scheduler (OS) is responsible to select an eligible flow (cross-point buffer) from the column and forward the existing packet(s) to the network interface. It also generates the corresponding credit (the size of the outgoing packet) and sends that to the associated credit scheduler (CS) of the input.

For every cross-point of the column the OS keeps the number of enqueued packets for each buffer in counters and based on these it generates an eligibility mask. When there are eligible flows in the column then the OS selects one of them by applying plain round robin policy and informs the network interface. The round robin policy is implemented with a priority enforcer which keeps state of the last served flow and it requires a single clock cycle for the decision. The cross-point logic needs 3 clock cycles due to synchronization to inform the OS for the packet and then the OS applies the scheduling policy and informs the network interface. This sequence of events requires only 5 clock cycles and therefore the OS can support cut-through operation even for minimum sized packets (24 bytes = 6 clock cycles). In order for OS to support back to back transmission to the network and to hide the scheduling latency it performs pre-scheduling. While a packet is transmitted and there are eligible flows, then the OS selects the next eligible flow according to the policy and informs the network interface 3 cycles before the previous packet finishes.

The packets are dequeued from the network interface when it has credits for the outgoing path while the cross-point selection of the OS is transparent to the interface. When the transmission starts then the OS informs the associated CS with the size of the packet.

3.2.4 Credit Scheduler (CS)

The credit schedulers (CS) are dedicated per input, follow the QFC-like approach [8] and hold the number of bytes that departed for a specific pair of input and output. Each CS receives from every OS the number of bytes that departed and originated from the associated input. The CS keeps state, in counters, of the accumulated bytes that departed whereas the wrap-around of the counters does not affect the protocol and the correctness of the system as described in [5],[8]. Note also that the OS and CS might work in different clock domains so the credit data are synchronized in 3 clock cycles.

The CS also interacts with the network interface and provides the credit data to be transmitted when requested; one credit at a time. However, CS is responsible to provide credits for every possible destination/output the input sends the packets. Therefore, the CS provides the credits to the network interface in a round robin fashion. The round robin policy is applied initially at the modified credits values and later at the unmodified to compensate for possible credit corruptions or losses.

3.2.5 RocketIO Network Interface

Each RocketIO network interface is responsible to decode the packet headers and enqueue the incoming packets in the appropriate cross-points of the associated crossbar row. It also keeps state of the incoming credits that concern the incoming buffer of the connected NIC. Besides, the network interface should transmit the outgoing packets and the credits it receives from the associated OS and CS whenever it has credits.

3.3 Data Packet Format

The data to be exchanged with the NetIF module are formed into packets. Figure 10 will guide us through this description. Each packet is formed from 68-bit words at the interface. These words exchanged to and from the NetIF module include 64 MS bits for the actual data, and the remaining 4 MS bits of each word add tagging (flags) and are only used to enhance the communication of the two modules (SOP, EOP, RFU). The data includes header and body (payload).The packet starts with a 68-bit (one word) header. The format of the actual header (the 64 bits) is shown in Figure 11. The 32 LS bits include the PCI address. The next 10 bits correspond to the packet size. The size is measured in words (64 bit words), and only the payload (the useful data -or body-) are measured. The header is thus excluded. The next 10 bits are Reserved for Future Use (RFU). Then, bits 51 through 55 form the opcode, something of no relevance to the NetIF. This will just be transmitted and will be used by the receiving side's PCI module. The next 7 bits carry the flow ID, which must be taken into account by the NetIF. The bit 63 is a flag that tells if the word is a header or a credit. 0 is used for header, 1 for a credit. Finally the 4 MS bits (only for the interface - not shown) carry the flags. For the header bit 64 bit must be set to 1 to signal SOP and the rest three to 0. For the other words all four must be zero, excluding the last word which must have bit 65 set to 1 to signal EOP.

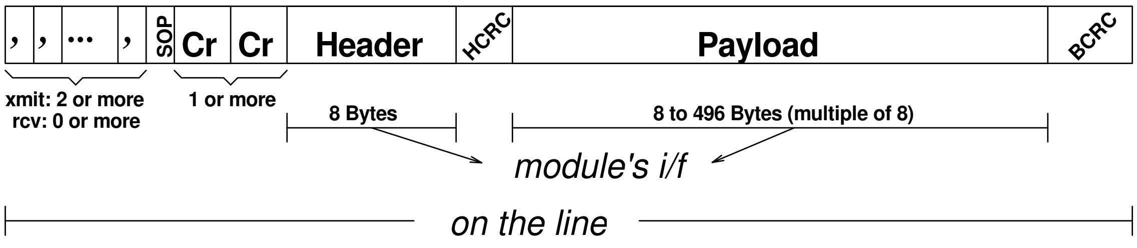

Figure 10: Packet format.

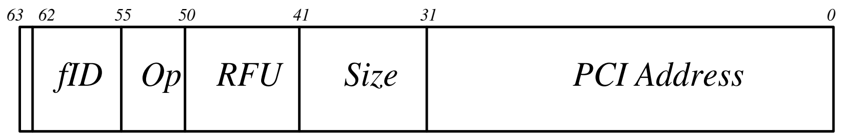

Figure 11: Packet header format.

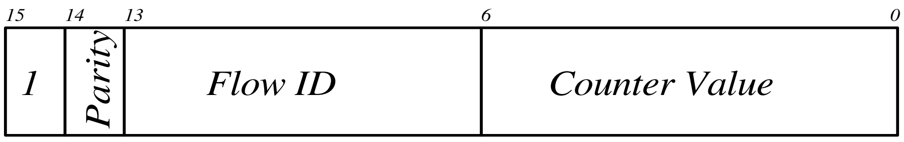

Figure 12: Credit format.

3.3.1 Credit Format and Credit Protocol

Figure 12 depicts the **credit format**. Each credit is 16 bits wide. The first 7 bits concern the credit value. How this value is calculated will be explained shortly. The next 7 bits describe the flow ID, which means the flow that this credit is meant for. With 7 bits we can support as much as 128 different flows. Bit 14 is the parity, which is the xor of the previous 14 bits (bits [13:0]). This is created at the transmitter and checked at receiver. If it is found to be wrong, the credit is discarded, i.e. it is not taken into account. Finally the MS bit, bit 15, is the flag that differentiates between a credit and a header, and should be 1 to designate a credit.

Now let us see the **credit protocol** in some detail. This way we will also understand the meaning of the "Counter Value" field, seen in Figure 12. The protocol is based on two cumulative counters at each NIC. One of these is counting the complete number of words that have been transmitted to the network since power up. The other counter keeps track of the number of words that have been forwarded from the network to the host, again in a cumulative manner since power up. This second counter is transmitted through the "Counter Value" of the credit to the other side. When the other side receives this credit, it has to compare it with its first counter (the one counting transmitted packets to the network) and see if there is enough space to send a new packet with a known size (the packet size is known through its header). In order for this protocol to work, correct initialization of the counters is needed. That is the fifo size at the other end must be taken into account, so that at start up this is the free space downstream. One other thing to take into account, is that a cumulative counter will sometime overflow. In order to compensate for this, the comparators at each side must be aware of value wrap around. Two extra bits for each counter are enough to implement this mechanism. A last issue concerns the width of the counters. Based on the RTT of the system, and the need for wrap around support, our counters end up being 12 bits wide. So the "Counter Value" of the credit is a subfield of the corresponding counter. Only the 7 most significant bits (bits [11:5]) are transmitted through the credit. The remaining 5 LS bits are not transmitted, and the receiver treats them as being all zero. This gives as a more coarse grained granularity for the credits, as an updated credit at the receiver will only be seen for every 32 (2^5) words transmitted. This however does not create any trouble.

4. VALIDATION AND PERFORMANCE RESULTS

4.1 NIC Validation

As part of our research, in parallel with the hardware prototype implementation we have also built a prototype I/O stack in the Linux kernel that makes use of the NIC prototype. In validating the NIC design we have so far used a setup of two nodes connected directly. The benchmarks we have used for validating our design are:

- Various user-level micro-benchmarks that access directly the NIC control registers for generating traffic. The micro-benchmarks use special driver calls to map the NIC control registers to program address space.
- The xdd micro-benchmark [11] with the *-dio* option to bypass the initiator's buffer cache. We vary the request size from 4 KBytes up to 1 MBytes, for both read and write accesses. Each experiment transfers a total of 4 GBytes between the initiator and the target.
- Filesystem-based experiments, where a filesystem is built on top of remote block device and the following operations are performed:
  - Create filesystem on top of the block storage volume, using the mkfs command-line utility. We build a filesystem of ReiserFS, with block-size equals to 4 KBytes.
  - Mount the filesystem, using the mount command-line utility.
  - Copy a compressed tar archive file from the dedicated IDE system disk to the filesystem. Specifically, we copy the source tree of the Linux kernel version 2.4.30, a tar file compressed using bzip2. The file size is 29.7 MBytes.
  - Extract the files from the compressed tar file. This step produces a 184.9 MBytes directory/file tree.
  - Create a tar archive file that contains the directories and files extracted in the previous step.
  - Compress the tar archive file produced in the previous step, using the bzip2 command-line utility.
  - Recursively remove all directories and files in the filesystem.
  - Un-mount the filesystem, using the umount command-line utility.
- A sequence of filesystem operations that manipulate a 515.4-MByte trace-file:
  - Copy the compressed trace-file from the dedicated IDE system disk to the filesystem.
  - Uncompress the trace-file, using the bunzip2 command-line utility.
  - Scan the trace-file, one-byte-at-a-time, using the wc command-line utility. The trace-file is in ASCII format, and contains 2,169,400 lines, each consisting of approximately 250 characters.
  - Compress the trace-file produced in the previous step, using the bzip2 command-line utility. This step produces a file of size 23.16 MBytes.
- A sequence of filesystem operations that result from the execution of the Postmark benchmark [12]. PostMark creates a set of files with random sizes within a configurable range, and then performs a number of transactions, consisting of a randomly-chosen pairing of file creation or deletion with file read or append. We setup Postmark to perform 5000 transactions, over a set of 1000 files, with sizes varying from 10 KBytes to 10 MBytes, no bias toward read/write transactions, and the read/write block size set to 4 KBytes. With these settings, Postmark reads 15.2 GBytes of data, and writes 21.6 GBytes.

4.2 Switch Validation

For the verification of the switch we developed software modules that initiate network traffic from the NICs to the switch with programmable packet sizes and programmable network destinations. These modules also examine the received and transmitted packets and report CRC error, packet losses or other system errors. We have run extensive tests on the system to verify the correct operation of the switch. The tests we have run are the following:

- *Ping-pong test between two NICs through the switch*: One of the machines is the initiator of the test which transmits a packet and waits the other machine, the target, to respond with the same packet. We have run this test with many different packet sizes in the range of 24bytes – 512bytes. We have made billions of iterations and overnight tests involving billions of packets to ensure the correct operation. Our system has not reported errors; hence the system passed successfully these tests.
- *1-way test through the switch:* One of the machines is the initiator of the test which transmits back to back packets to a target machine through the switch without waiting any response. We have run this test with many different packet sizes in the range of 24bytes – 512bytes. We have made billions of iterations and overnight tests involving billions of packets to ensure the correct operation. Our system has not reported errors; hence the system passed successfully these tests. The measured throughput was 287 Mbytes/sec (theoretical max. 300 Mbytes/sec).
- *1-way test to itself through the switch*: The initiator of the test transmits back to back packets to itself through the switch without waiting any response. We have run this test with many different packet sizes in the range of 24bytes – 512bytes.We have made billions of iterations and overnight tests involving billions of packets to ensure the correct operation. Our system has not reported errors; hence the system passed successfully these tests. The measured throughput was 270 Mbytes/sec since there is both incoming and outgoing traffic to the NIC (theoretical max. 300 Mbytes/sec).
- *1-way test with 3 senders and 1 receiver:* There are 3 initiators that transmit back to back packets to 1 single receiver through the switch without waiting any response. This test stresses out the switch because is generates output contention and activates the flow control mechanisms of the NICs and the switch. We have run this test with many different packet sizes in the range of 24bytes – 512bytes.We have made billions of iterations and overnight tests involving billions of packets to ensure the correct operation. Our system has not reported errors; hence the system passed successfully these tests. The measured throughput was 93 Mbytes/sec on every sender, hence 279 Mbytes/sec on the receiver (theoretical max. 300 Mbytes/sec).
- *Round robin 1-way test to any network destination:* There are 4 initiators that transmit in round robin fashion to all the network destinations without waiting any response. The round robin policy is applied in a per packet basis. This and all the other tests have fair temporal randomness since the network nodes obtain their packet data through PCI-X DMA accesses which come under the arbitration policy of each host's PCI-X bridge. This test also stresses out the switch because is generates output contention and activates the flow control mechanisms of the NICs and the switch. We have run this test with many different packet sizes in the range of 24bytes – 512bytes.We have made billions of iterations and overnight tests involving billions of packets to ensure the correct operation. Our system has not reported errors; hence the system passed successfully these tests.

The next step, as part of future work, is to accurately measure the metrics that indicate the buffered crossbar performance under balanced or unbalanced traffic, with more random network destination and packet sizes.

4.3 Preliminary Performance Results

Figure 13 shows the throughput for a simple, user-level benchmark on top of two nodes connected directly (without the switch), using a pair of RocketIO links. The prototype we use in our experiments consists of two Dell 1600SC servers, each with a single Intel Xeon CPU, running at 2.4 GHz, 512 MBytes of main memory, and two 64-bit PCI-X buses running at 100 MHz. The two nodes have a dedicated IDE system disk, and two types of interconnects: (a) a Gigabit Ethernet adapter for system administration and monitoring and (b) our custom NIC for all data transfers. The theoretical maximum throughput of the host-NIC DMA engine is one 64-bit word at every 100-MHz PCI-X clock cycle or 762.9 MBytes/s (assuming zero bus arbitration and protocol overheads). The theoretical maximum throughput for the pair of RocketI/O links is one 64-bit word at every 78.125-MHz Rocket-I/O clock cycle or 596 MBytes/s. Therefore, the maximum theoretical end-to-end throughput is limited to that of the network links, i.e. 596 MBytes/s.

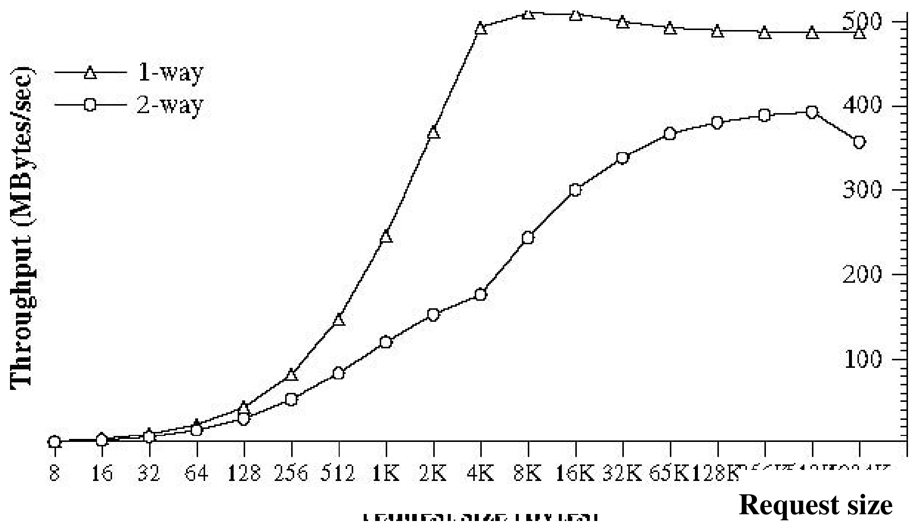

Figure 13: Throughput for two nodes connected directly, for a one-way (1-way) and a ping-pong test (2-way).

5. CONCLUSIONS

In this work we present the detailed implementation of a multi-gigabit NIC and a scalable buffered crossbar switch. The implementation is based on an FPGA-based prototyping platform. The prototypes have been implemented and validated using both micro-benchmarks as well as a real I/O protocol stack in a 4-node setup. The prototypes are aimed to support research in high-speed interconnects. The current prototypes include features found in state-of-the-art systems today and are currently used for examining issues in supporting higher network speeds using multiple physical links, new ordering semantics, and flow-control mechanisms.

6. ACKNOWLEDGEMENTS

We would like to thank the members of ICS-FORTH that have participated in numerous useful discussions. We thankfully acknowledge the support of the European FP6-IST program through the SIVSS (STREP 002075) and UNIsIX (MC EXT 509595) projects and the HiPEAC Network of Excellence (NoE 004408).

7. REFERENCES

[1] D. Stephens, Hui Zhang: "Implementing Distributed Packet Fair Queueing in a Scalable Switch Architecture", IEEE INFOCOM'98 Conference.
[2] F. Abel, C. Minkenberg, R. Luijten, M. Gusat, I. Iliadis: "A Four-Terabit Packet Switch Supporting Long Round-Trip Times", IEEE Micro Magazine, Jan./Feb. 2003, pp. 10-24.
[3] N. Chrysos, M. Katevenis: "Weighted Fairness in Buffered Crossbar Scheduling", Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR 2003), Torino, Italy, June 2003, pp. 17-22; http://archvlsi.ics.forth.gr/bufxbar/bxb_scheduling.html
[4] G. Sapountzis, M. Katevenis: "Benes Switching Fabrics with O(N)-Complexity Internal Backpressure", Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR 2003), Torino, Italy, June 2003, pp. 11-16; http://archvlsi.ics.forth.gr/bpbenes/
[5] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, N. Chrysos: "Variable Packet Size Buffered Crossbar (CICQ) Switches", *Proc. IEEE International Conference on Communications (ICC 2004)*, Paris, France, 20-24 June 2004, vol. 2, pp. 1090-1096.
[6] M. Katevenis, G. Passas: "Variable-Size Multipacket Segments in Buffered Crossbar (CICQ) Architectures", *Proc. IEEE International Conference on Communications (ICC 2005)*, Seoul, Korea, 16-20 May 2005, CR-ROM paper ID "09GC08-4", 6 pages.

[7] N. Chrysos, M. Katevenis: "Multiple Priorities in a Two-Lane Buffered Crossbar", *Proc. IEEE Globecom 2004 Conference*, Dallas, TX, USA, 29 Nov. - 4 Dec. 2004, CR-ROM paper ID "GE15-3", 7 pages;

[8] D. Simos: "Design of a 32x32 Variable-Packet-Size Buffered Crossbar Switch Chip", *Technical Report FORTH-ICS/TR-339*, Inst. of Computer Science, FORTH, Heraklion, Crete, Greece; M.Sc. Thesis, Univ. of Crete; July 2004, 102 pages.

[9] Xilinx ML325 Characterization Board. URL: http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=HW-V2P-ML325&sGlobalNavPick=PRODUCTS&sSecondaryNavPick=BOARDS.

[10] DiniGroup DN6000K10SC Development Board. URL: http://www.dinigroup.com/DN6000k10SC.php.

[11] I/O Performance Inc. The xdd i/o benchmark. http://www.ioperformance.com.

[12] J. Katcher. Postmark: A new file system benchmark, 1997. Technical Report TR3022, Network Applicance Inc.